Technische Universität München
Fakultät für Informatik
Dr. Tobias Lasser

WS 2016/17
Exercise Sheet 8
January 17, 2017

Exercises in Image Processing and Visualization using C++

This assignment represents the third and final part of your project. After you have set up your X-Ray simulated acquisition in the first part and reconstruction in the second part, you will now visualize your results in order to get insight into the acquired data. At the end of this assignment, you should have a basic visualization tool that allows you to inspect the *mystery data* so that you can draw conclusions on what is inside.

Exercise 1 Getting started with the basics

For this assignment, you will need a Qt Widget that shows both the widgets to setup/control the visualization as well as the visualization itself (similar to your image editor from assignment 5). Figure 1 shows an exemplary implementation of ours. The left column holds all widgets to load a data set and setup the rendering. The center column shows a 2D MPR visualization. The right column shows a Maximum Intensity Projection of the data set.

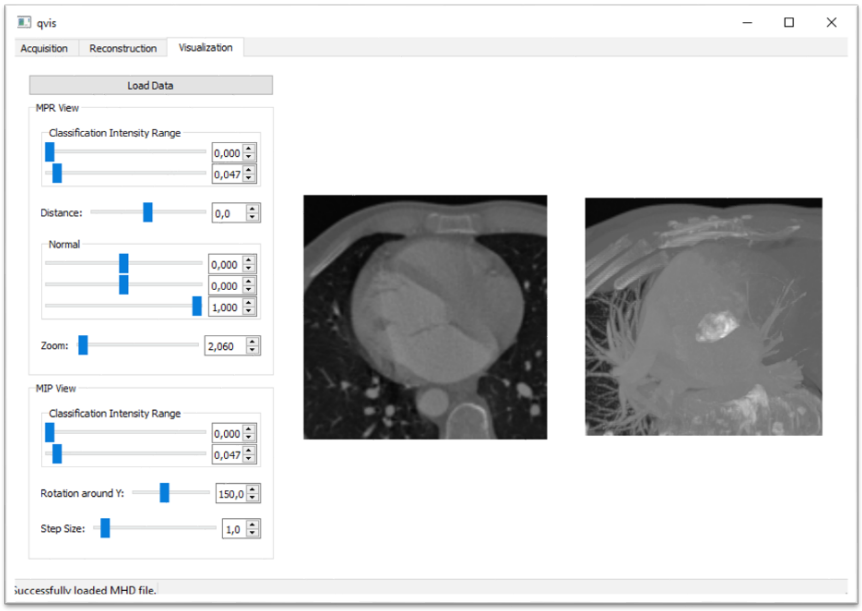

Figure 1: Screenshot of our example implementation visualizing a human thorax CT data set.

Think about how a good software (class) design would look like to implement the following functionality:

- Storing and accessing volumetric images (you should already have this)
- Performing classification on volume intensities
- 2D MPR visualization of the volume using arbitrary oriented planes
- 3D volume visualization using ray casting and a simple compositing scheme

The following exercises will guide you through the process of implementation.

Exercise 2 Accessing the volume data

In the previous assignment, we introduced the EDF file format to store volumetric images. Implement functionality to directly load your reconstruction result from assignment 7 into the visualization module. Additionally, you might want to be able to load a EDF file directly for debugging and reviewing external volumes.

You already designed a C++ class to store the volume and access its contents. If not done already, extend this class with a method `getVoxel(int x, int y, int z)` to access single voxels using integer parameters. Furthermore, implement a method `getVoxelLinear(float x, float y, float z)` for trilinear interpolation (as discussed in the lecture) to access the volume data also "in between" the voxels using float parameters. Make sure to handle out-of-bounds access correctly!

Exercise 3 Classification

In the lecture, we introduced the concept of classification and transfer functions for converting volume intensities to optical properties, such as color and opacity. Implement this functionality into your application. For instance, implement a class `TransferFunction` holding the TF parameters and having a method `QColor classify(float intensity)`.

Implement a simple linear ramp transfer function (sometimes also called *windowing*), which is specified by an intensity range $[i_{\min}, i_{\max}]$ and a color (r,g,b,a). The left end of the intensity range ($i_{\min}$) specifies the intensity where a corresponding voxel is classified with the given color at 0% opacity $(r,g,b,0)$. The right end of the intensity range ($i_{\max}$) specifies the intensity where a corresponding voxel is classified with the given color at 100% opacity $(r,g,b,255)$. All intensities in between are classified using linear interpolation. All intensities outside the intensity range are mapped to transparent or fully opaque color.

You are welcome to implement more complex transfer functions, for instance using an arbitrary amount of key points/colors. However, this is fully optional and only to get visually more appealing results.

Add appropriate user interface widgets to change the parametrization of the linear ramp. If you choose to implement more advanced transfer functions, you do not have to provide a complete set of widgets but at least provide an option to switch between the ramp and the advanced transfer function.

Exercise 4 2D MPR Visualization

Implement a 2D multi-planar reconstruction (MPR) visualization for your application. As presented in the lecture, such a visualization is defined by a 2D cut plane that intersects with the volume. This plane can be parameterized in many different ways. We suggest to use the Hesse Normal Form[1] (HNF), since it has many useful properties.

Implement a custom `QWidget` and override the `paintEvent(QPaintEvent*)` method to implement your visualization. Find a transformation from your widget's pixel space to the parameterization space of the 2D MPR plane (when using the HNF, the cross product will be a good tool for that). For each pixel of your widget, compute the corresponding voxel, use trilinear interpolation to get the local intensity and then apply your transfer function to get the optical properties. Finally, paint the pixel in the computed color to yield the 2D MPR visualization.

Since the volumes that we use for our seminar are very small, we also need a zoom functionality to zoom in and out. In order to get the best possible result, directly include the zoom factor into the transformation from pixel space to parameterization space.

[1] `https://en.wikipedia.org/wiki/Hesse_normal_form`

Also provide a suitable user interface to arbitrarily change the position and orientation of the MPR plane.

Hint: You may experience a rather slow rendering performance. Before you waste time with premature optimization[2], consult the Qt manual on painting/rendering efficiency. Most probably, your implementation forces a high amount of locks/state changes in the Qt rendering engine.

Exercise 5 Direct Volume Rendering

Implement a simple 3D volume visualization using ray casting. As discussed in the lecture, such a visualization is defined by three things:

- Camera/projection setup (defining a set of rays)
- Ray discretization (step size)
- Compositing mode (e.g. MIP, DVR, . . .)

Again, implement a custom `QWidget` and override the `paintEvent(QPaintEvent*)` method to implement your visualization. The easiest ray setup would be to use an orthographic projection that can rotate around one axis of the volume. If you do the Math correctly, this allows you to easily compute the entry and exit points for each ray directly (the points where the ray hits the target volume). Discretize each ray by a user-defined step size and use trilinear interpolation to collect the sample intensities along each ray. Use Maximum Intensity Projection (MIP) as compositing mode and apply the transfer function to compute the final pixel color.

Important: In contrast to the voxel traversal ray casting algorithm you implemented in the previous assignment for the CT reconstruction, this time you will implement discrete ray casting with a fixed step size. Hence, given a ray direction and start/entry point, you will advance along the ray by a given length (step size). Every step you will collect a voxel intensity at the corresponding position using trilinear interpolation.

If you want to go fancier (again: fully optional), you can also implement a customizable perspective projection[3] for better visual results. Traditionally, the trackball metaphor [4] is used to rotate a view around 3D objects. You are welcome to implement this, but a collection of buttons/sliders to control the orientation and zoom of the volume is completely sufficient for this exercise. If you implemented advanced transfer functions in Exercise 3, you may want to implement Direct Volume Rendering (DVR) as compositing scheme as well.

Hint: Ray casting is a computationally expensive task. Thus, restrict your visualization to a small viewport (e.g. 200x200 pixels) in order to get an acceptable rendering performance. Parallelization frameworks such as OpenMP or Intel TBB may help you distributing the work load over all your CPU cores.

[2] `http://c2.com/cgi/wiki?PrematureOptimization`

[3] Shirley et al.: Fundamentals of Computer Graphics, Chapter 4 (A.K. Peters)

[4] `https://www.opengl.org/wiki/Object_Mouse_Trackball`